StegWare: A Novel Malware Model Exploiting Payload Steganography and Dynamic Compilation

Daniele Albanese[1], Rosangela Casolare[2], Giovanni Ciaramella[1], Giacomo Iadarola[1], Fabio Martinelli[1], Francesco Mercaldo[1,2], Marco Russodivito[2] and Antonella Santone[2]
[1]*Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy*
[2]*University of Molise, Campobasso, Italy*

Keywords: Security, Malware, Android, Reflection, Dynamic Compiling, Dynamic Loading, Steganography.

Abstract: Android is the most widely used mobile operating system in the world. Due to its popularity, has become a target for attackers who are constantly working to develop aggressive malicious payloads aimed to steal confidential and sensitive data from our mobile devices. Despite the security policies provided by the Android operating system, malicious applications continue to proliferate on official and third-party markets. Unfortunately, current anti-malware software is unable to detect the so-called zero-day threats due to its signature-based approach. For this reason, it is necessary to develop methods aimed to enforce Android security mechanisms. With this in mind, in this paper we highlight how a series of features available in current high-level programming languages and typically used for totally legitimate purposes, can become a potential source of malicious payload injection if used in a given sequence. To demonstrate the effectiveness to perpetrate this attack, we design a new malware model that takes advantage of several Android features inherited from the Java language, such as reflection, dynamic compilation, and dynamic loading including steganographic techniques to hide the malicious payload code. We implement the proposed malware model in the *Stegware* Android application. In detail, the proposed malware model is based, on the app side, on the compilation and execution of Java code at runtime and, from the attacker side, on a software architecture capable of making the new malware model automatic and distributed. We evaluate the effectiveness of the proposed malware model by submitting it to 73 free and commercial antimalware, and by demonstrating its ability to circumvent the security features of the Android operating systems and the current antimalware detection.

1 INTRODUCTION

Android is nowadays the most diffused operating system on mobile devices. As a matter of fact, in a report conducted by StatCounter in October 2022, the number of devices where Android is running is estimated at over 80%[1]. The usage of this operating system is not only restricted to mobile devices but is largely considered in automotive (Mercaldo. et al., 2022) and IoT fields. Because of its widespread distribution, attackers are writing increasingly aggressive malicious payloads to obtain sensitive and private data saved on our devices. Due to the Android open source nature, users could obtain mobile applications from third-party markets rather than the original Google Play Store, at more competitive, if not free, costs. Since these alternative markets are not managed by Google, attackers publish malware on these platforms misleading users to install on their devices spoofed applications. To limit these phenomena and to protect users over the years, researchers developed different security measures to protect users from possible attacks, for instance, based on machine learning or model checking(Scott, 2017). From the operating system side, Google introduced the usage of sandboxes, permission, and *Google Play Protect*. Sandboxes are used to provide a unique identification code to each program and operate it within a confined memory area or execute a limited number of system calls, whereas permissions are employed to govern the access to information by apps served by users. In 2012 was launched Google Play Protect, i.e., a tool

[1]https://gs.statcounter.com/os-market-share/mobile/worldwide/

Albanese, D., Casolare, R., Ciaramella, G., Iadarola, G., Martinelli, F., Mercaldo, F., Russodivito, M. and Santone, A.
StegWare: A Novel Malware Model Exploiting Payload Steganography and Dynamic Compilation.
DOI: 10.5220/0011859000003405
In *Proceedings of the 9th International Conference on Information Systems Security and Privacy (ICISSP 2023)*, pages 741-748
ISBN: 978-989-758-624-8; ISSN: 2184-4356

for detecting malware and trojans. But these countermeasures are not enough: as a matter of fact, according to a McAfee report issued in 2022 [2], cybercriminals are always active in trying to defraud mobile users. In addition to the more deceptive phishing approach, a new way has just been devised to mislead mobile game cheaters by adding dangerous code to an existing open-source game hacking tool. Following a survey released by *financesonline.com* [3], there is a not-specified population between the ages of 6 and 15 who spend time gaming on mobile devices with the permission of their parents. This element makes it considerably easier for attackers to hit their intended target. From these considerations it emerges the need to develop new ways to protect the personal and sensitive user's mobile information, by boosting researchers, from both industrial and academic sides, to focus on a new way to perpetrate attacks on the mobile environment to develop more security features by anticipating malware writers. With this in mind, in this paper, we introduce a novel malware model able to overcome the security mechanisms provided by Android operating systems and the free and commercial antimalware detection approach. In the proposed model the malicious payload is delivered through an image where the code is hidden through steganography. Once the payload source code is retrieved is automatically compiled with dynamic compilation, thus loaded into memory through dynamic loading and invoked at run-time by exploiting reflection, a mechanism provided by Android and modern object-oriented programming languages.

Our model relies on the combined exploitation of three mechanisms native provided by the Android programming language: dynamic compiling, reflection, and dynamic loading, to allow a series of source code snippets to combine into a running application and execute, to dynamically alter the normal flow of program execution. Moreover, the source code snippets are hidden into images by exploiting steganography (Johnson and Jajodia, 1998). The most popular steganography technique, which is usually used with picture and sound carrier files, is known as Least Significant Bit Substitution (LSBS) or overwriting. This method consists of overwriting the bit with the lowest arithmetic value going to modify the original output slightly enough to be unlikely to be detected by human senses (Siper et al., 2005). Although the LSBS technique may turn out to be efficient, modern steganography applications change the last bit randomly. The latter is performed to obstacle adversaries.

We implemented the new attack model we propose into the *StegWare* Android malware, to demonstrate the possibility to perform this kind of attack in a real-world environment.

The remaining of the paper proceeds as follows: in Section 2 we present the novel malware model; in Section 3 we discuss the *StegWare* implementation; the *StegWare* experimental analysis is presented in Section 4; in Section 5 we report the current state-of-art literature on the dynamic loading and dynamic compilation adoption for malicious purposes and, finally, in the last section conclusions and future research plans are drawn.

2 THE MALWARE MODEL

In this section, we describe the proposed approach behind the proposed novel malware model. In Figure 1 we show the malware architecture, which is explained in detail below.

In the proposed novel malware model, we consider a scenario where an image is delivered to the users, for instance through the browser or an instant messaging application such as WhatsApp or Telegram. Once the image is received and stored into the Android application, the following steps will be activated, as shown in Figure 1:

1. *Payload Search:* to search for malicious images, a service has been implemented that analyzes all the multimedia files in the memory of the device, searching for a specific file with a specific name. Thus the malware model is continuously and actively looking for images from different sources;
2. *Payload Extraction:* once an image is gathered, the model tries to extract the source code from the image (whether available). For image decoding, an *ImageSteganography* object is instantiated to which a bitmap image is given as input. Then, another object is instantiated, but this time of type *TextDecoding*, to which the *TextDecodingCallback* parameter is given as input. After these operations, the decoding task of the *TextDecoding* object will be executed on the *ImageSteganography* object and, after the *override* operation of the *onCompleteTextDecoding* method, it will be possible to see the Java code hidden inside the image passed as input;
3. *Payload Execution:* using the output of the previous step, the malicious Java code hidden inside the image will be processed by the approach in

[2] https://www.mcafee.com/blogs/mobile-security/mcafee-2022-consumer-mobile-threat-report/

[3] https://financesonline.com/mobile-gaming-demographics/

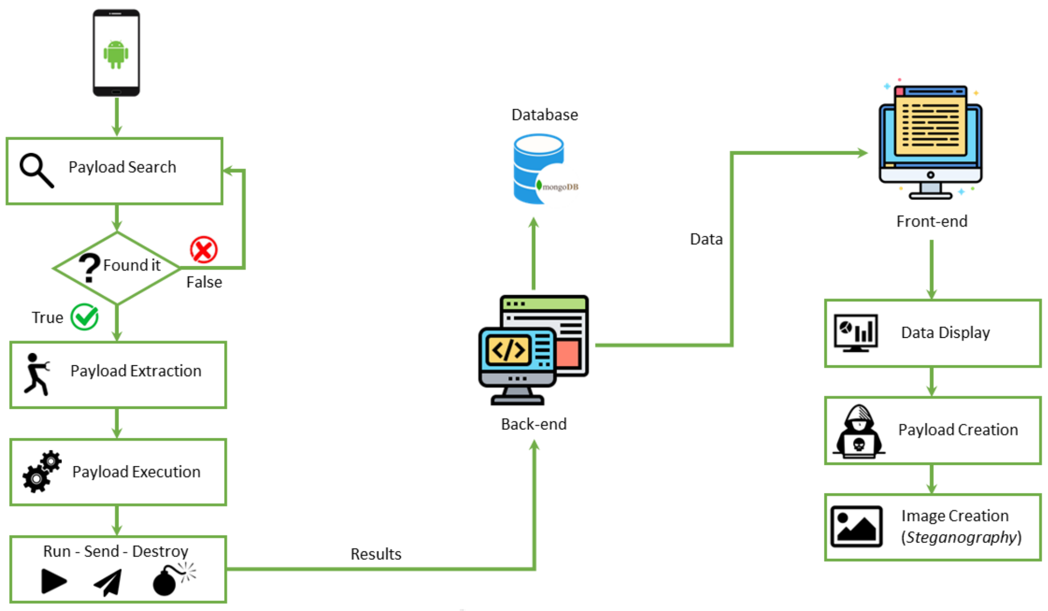


Figure 1: High-level architecture scheme of malware model.

Figure 1:

- the payload is scanned to understand if it is a valid Java class;
- the Java class is processed for building the Abstract Syntax Tree (*AST*);
- the *AST* will be mapped using the JavaAssist API[4] to generate the compiled bytecode of the Java class and it will be written into a `.class` file;
- the `.class` file will be packed into a `.dex` i.e., a Dalvik executable (the form of executable code used in Android applications);
- the `.dex` file, we can load it into the Android device memory using the `dalvik.system.DexClassLoader` API;

4. *Payload Run:* when the bytecode is completely loaded into the Android device RAM, we can use `java.lang.reflect` API to instantiate the class that we have previously compiled;
5. *Payload Send:* once the code has been extracted and executed, a call will be made to the *backend* which will contain all the data that the payload was able to extract;
6. *Payload Destroy:* once executed, we need to delete all the traces left by the malware to restore the original application state, making the proposed model stealth; In fact during the dynamic compiling steps, the model generated a `.class` and a `.dex` into the Android device storage. The malicious code should be in the application only within the dynamic loading and execution time window and then these files must be permanently deleted from the device.

[4]https://www.javassist.org/

As discussed the proposed malware model considers several characteristics inherited from the Android programming language, by considering also the ability to generate images that contain hidden code inside them with the adoption of steganography. This operation is performed through the *frontend* and using the *steganography*, aimed to hide message fragments within a bit vector.

In detail, we adopted the *LSB* steganography: this technique is mainly based on the concept that the appearance of a high-definition digital image does not change if the colors are subtly modified. In images, each pixel is represented by a different color, and by changing the last bit of each pixel value, the color will not be significantly changed and the image content will be preserved despite this manipulation. This operation causes a person to be unable to tell with the naked eye whether there has been an image manipulation operation.

Considering the use that users make of smartphones every day, it becomes very easy to get hold of images that can be potentially harmful. Suffice it to say that every day a large number of photos and images are sent and received via messaging applications (for instance, Telegram or WhatsApp), in addition to images that can be downloaded from websites via the browsers installed on the devices.

3 THE StegWare MALWARE

In this section, we describe the technologies we exploited to implement the novel malware model into the Stegware Android application. In particular, we explain how we built the back end using Node.js, the front end using React JS, and the not-relational database we considered i.e., Mongo DB. The source code we developed for the app [5], dashboard [6], and back end [7] is freely available for research purposes on the GitHub platform. In the following, we present the implementation related to the *Stegware* application, the back-end, the front-end, and the non-relational database we exploited.

3.1 The *StegWare* Android App

The Android application was developed using the Android native language: it is composed of a unique *Main Activity*, shown in Figure 2, where the project's logo is reported. Once the app has been loaded into the victim device, the user needs to accept the permissions required, i.e., read and write on the device's storage. After the acceptance, the *StegWare* application starts the Communication Service through the *START STICKY* modality. The latter allows the Service to remain in execution in the background. In a loop, the Communication Service attempts to detect harmful pictures during execution. If the image is recognized, the service decodes it using steganography. This operation is performed to retrieve the malicious payload. When the latter is obtained, dynamic compilation, dynamic load, and reflection execution begin. In the end, an API request to the server is sent.

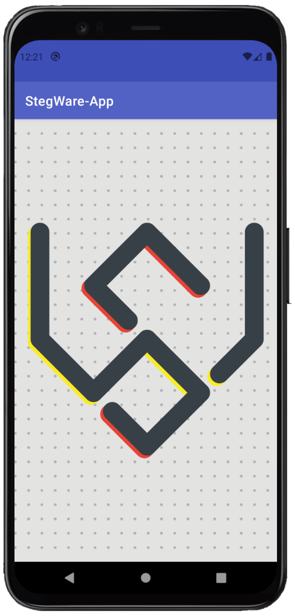


Figure 2: *StegWare* Main Activity.

3.2 Back-End

To develop the back-end our choice fell on Node.js [8], thanks to its versatility, the large number of libraries, and the high performance and management obtained on requests. To start this component we also employed Docker as container.

The back end aims to handle communication between the database and the Android application through the use of the end-point to allow the external client to invoke the APIs. Deeper, the end-point management was entrusted to the framework Express [9]. The latter provides some advanced methods for handling HTTP requests and for implementing APIs. In Listing 1 we reported the setup of server Node.js using Express.

```
// Dependencies
const http = require('http');
const express = require('express');
const bodyParser = require('body-parser');

// Server port
const PORT = 9999;

// Create a new express application
const app = express();

// Middleware - body-parser config
app.use(bodyParser.json());
app.use(bodyParser,urlencoded({ extended: true
    }));

// Create a new HTTP server
const server = http.createServer(app);

// Start server
server.listen(PORT);
```

Listing 1: Setup ExpressJS framework.

3.3 Front-End

To simplify malware management we also built a dashboard using the React.js framework to help with malware management. Furthermore, the graphics component was created with the MaterialUI library, while using the Axios framework HTTPS request are managed. Users may view all of the possible payloads that can be sent to an Android smartphone via

[5]https://github.com/dj-d/StegWare-App
[6]https://github.com/dj-d/StegWare-Dashboard
[7]https://github.com/dj-d/StegWare-Backend
[8]https://nodejs.org/en/
[9]https://expressjs.com/

the dashboard. In Listing 2 a payload example is reported. People may also alter the payload using an editor built with the Monaco React framework, add a new payload, or delete it, as shown in Figure 3. The front-end interface also allows users to recover past assaults and inspect individual attack data, as illustrated in Figure 4.

```
  import android.content.Context;
    import android.location.Location;
    import android.location.LocationManager;
    class RuntimeClasse {
    public RuntimeClasse() {}
    public String run(Context context) {
    LocationManager locationManager =
         (LocationManager)
    context.getSystemService \
       \ (Context.LOCATION_SERVICE);
    Location location =
         locationManager.getLastKnownLocation \
       \ ("network");
    return location.toString();
    }
}
```

Listing 2: Payload to retrieve last victim's position.

3.4 Database

As a database, we chose a non-rational database, i.e., Mongo DB [10]. The reason we chose this form of the database may be discovered in the data processing structure, which is not defined a priori. Also the Mongo DB installation happened through a Docker image.

4 EXPERIMENTAL ANALYSIS

In this section, we demonstrate the ability of the *SteWare* malware to elude the operating system's security features and to avoid detection from commercial and free antimalware. In particular, we propose two different experimental analyses: the first one is a static analysis i.e., it does not require running the application it consists in submitting the *StegWare* app to several free and commercial antimalware to understand whether the *StegWare* malicious behavior is detected. The second experimental analysis considers the installation of the *StegWare* prototype on a real-world environment, to understand whether the Android operating system blocks the malicious payload execution in some phases (for instance, in the source code recovering, the dynamic compiling or the dynamic loading). Different antimalware will also be installed on the device when installing and running *StegWare*, in particular, we selected 5 different well-known antimalware that were installed, one at a time, when *StegWare* was installed and run with the malicious payload execution: for this reason we consider this analysis as a dynamic one because it considers the *StegWare* execution. In the following, we explain in detail how we conducted and the results we obtained from both the static and dynamic analysis.

[10]https://www.mongodb.com/

4.1 The Static Analysis

To this aim, we submitted the *StegWare* prototype to two different webs antimalware aggregators: the first one is *VirusTotal*[11], while the second one is *Jotti*[12] i.e., two concurrent online scanning systems, which together use 73 antimalware.

VirusTotal is a free online scanning service that allows the detection of different types of malware present in suspicious URLs or files. These are subjected to the simultaneous static analysis of numerous antimalware constantly updated to the latest version available. Since 2012, VirusTotal has belonged to Google, which bought it, improved it, and then integrated it into the automated control procedures performed before the applications were published on the Google Play Store. Although this tool allows to speed up the first part of static analysis, it is always good to manually inspect the code for any false negatives.
The *StegWare* APK was checked by VirusTotal on the date 2021/12/01 and none of the 60 anti-malware reported anomalies.

The Virustotal report is freely available [13].

The second service we used is *Jotti*, which, in the same way as VirusTotal, allows us to perform the simultaneous analysis of suspicious files between different anti-malware.

The results of the scans are shared to improve the accuracy of scans. The *StegWare* APK was checked by Jotti on the date 2021/12/01 and none of the 13 anti-malware reported any anomalies.

4.2 The Dynamic Analysis

The idea behind this second experimental analysis is to understand if the Android operating system can intercept (someone of) the *StegWare* malicious behavior. Moreover, we want to understand whether antimalware can block the execution of *StegWare* malware.

[11]https://www.virustotal.com/gui/home/upload
[12]https://virusscan.jotti.org/it
[13]https://www.virustotal.com/gui/file/5ff2aac304df0293a8cbc7e55582acd3e6c20dc7b97986ac2bfa00f415cd8d9e

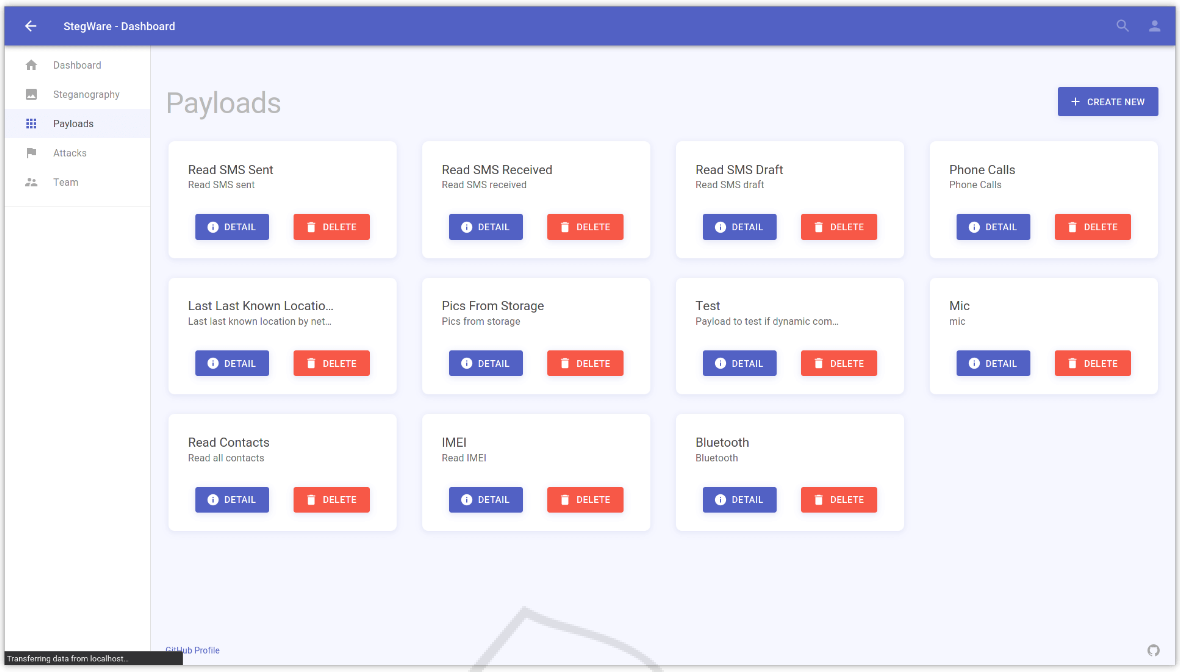


Figure 3: Dashboard Page - Payload.

This analysis consists of the installation and execution of the *StegWare* malware in a real-world device while the antimalware daemon and all the antimalware heuristics were previously activated. For this purpose, We install the *StegWare* malware on a physical device (i.e., a Samsung Galaxy S9 Plus with Android 10 on board) and we choose several best-ranked antimalware available for the Android environment.

In particular, we selected the five best antimalware from the ones present in the ranking drawn up by AV-TEST (an independent organization providing comparative antimalware tests and reviews)[14] in May 2020, which received full marks as regards the level of protection, performance, and usability.

The tests have been carried out with the following procedure:

1. the antimalware is installed;
2. the antimalware daemon is installed and enabled;
3. the antimalware heuristics are enabled;
4. the *StegWare* application is installed and initialised;
5. the attacker sends a malicious payload;
6. the expected behavior of *StegWare* is observed;
7. the attacker received the information gathered from the device;
8. whether there is the availability of another malicious payload the procedure goes to step 5;
9. the *StegWare* applications is uninstalled;
10. the antimalware is uninstalled;
11. a new antimalware is installed and the procedure goes to step 2.

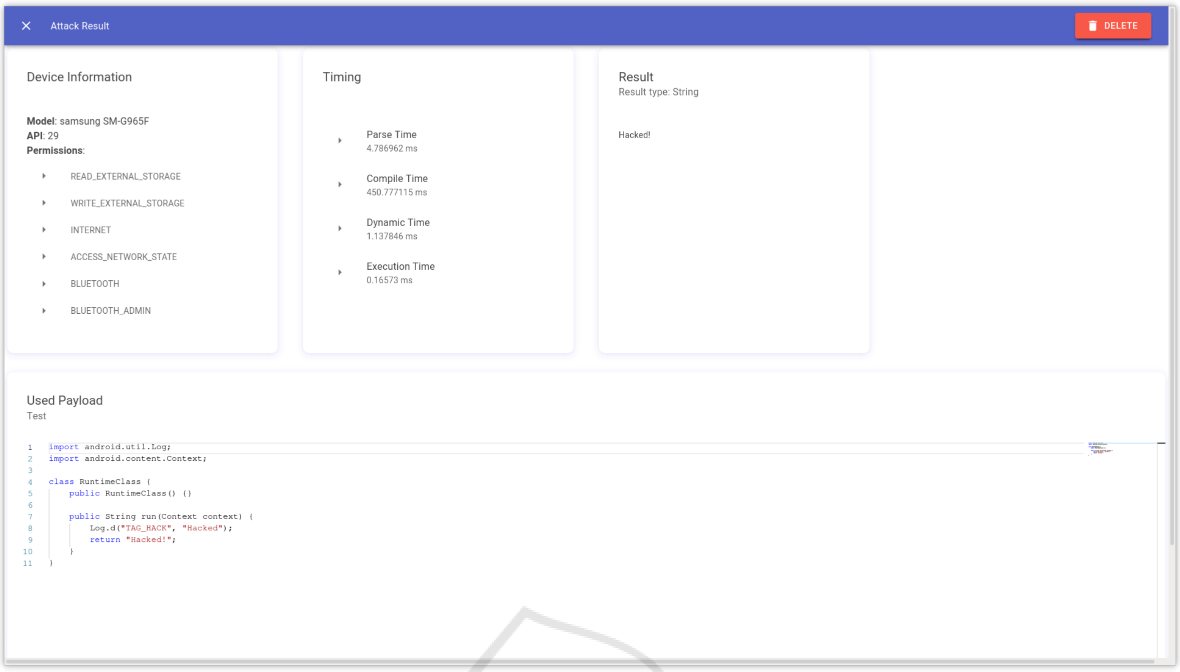
The following antimalware is considered in the in-depth analysis: Avira, BitDefender, GData, Kaspersky, and McAfee. We recall that the *StegWare* Android application was installed on the device and we carried out all the attacks: none of the antimalware detected anomalies at any stage of the proposed malware model.

This happens because currently antimalware considers the so-called signature-based mechanism i.e., the payload is successfully detected whether its signature is matching a signature stored in the database repository. Additionally, several antimalware exploits some heuristic scanning methods finalized to detect malware without needing a signature. This is why most antimalware programs use both signature and heuristic-based methods in combination, to catch any malware that may try to evade detection.

5 RELATED WORK

In the literature several works focus on dynamic loading and dynamic compilation, modifying the control

[14]https://www.av-test.org/en/



Device Information

Model: samsung SM-G965F
API: 29
Permissions:

- READ_EXTERNAL_STORAGE
- WRITE_EXTERNAL_STORAGE
- INTERNET
- ACCESS_NETWORK_STATE
- BLUETOOTH
- BLUETOOTH_ADMIN

Timing

- Parse Time 4.786962 ms
- Compile Time 450.777115 ms
- Dynamic Time 1.137846 ms
- Execution Time 0.16573 ms

Result
Result type: String

Hacked!

Used Payload
Test

```
import android.util.Log;
import android.content.Context;

class RuntimeClass {
    public RuntimeClass() {}

    public String run(Context context) {
        Log.d("TAG_HACK", "Hacked");
        return "Hacked!";
    }
}
```

Figure 4: Dashboard Page - Result.

flow of an application in a mobile environment.

Researchers in (Casolare et al., 2021) propose a new attack model called *2Faces* targeting the Android platform. This model is based on the combined exploitation of three native mechanisms provided by the Android programming language: dynamic compilation, reflection, and dynamic loading allowing a series of source code snippets to combine into a running application, to alter dynamically the normal flow of program execution. The main difference between the *StegWare* approach and the one proposed in *2Faces* relies on the adoption of the stenography, in particular the LSB (i.e., a steganography technique in which we hide messages inside an image by replacing Least significant bit of image with the bits of a message to be hidden), not exploited by the *2Faces* malware.

In (Canfora et al., 2015) the authors designed an Android malware model, called *Composition Malware*, which uses dynamic loading and reflection to execute external Dalvik byte code which is not present at installation time in the application. The model we propose can compile the payload source code at run-time, *Composition Malware* on the other hand is aimed to execute a pre-compiled executable file; therefore it requires that the executable code transits over the network (in clear) and for this reason, it could be detected by the network traffic analysis mechanisms. Differently in the malware model proposed by us, only encrypted source code snippets hidden in images transit over the network.

Authors in (Wang et al., 2013) worked on developing an application for the iOS operating system, aimed to dynamically load code not initially present in the application code as reviewed by Apple. Once installed, the application allows performing various malicious activities, such as posting tweets, sending e-mails and SMS, taking photos, and acquiring information on the identity of the device. The purpose is to demonstrate that it is also possible in Apple's operating system to dynamically load executable code at run-time. The *StegWare* model, in addition to dynamic loading, includes code reflection, and dynamic compiling with source code hiding employing steganography.

Researchers in (Prandini and Ramilli, 2012) studied the adoption of the Return-Oriented Programming (ROP), by altering the control flow at run-time, to propose an exploit technique that allows the attacker to have control of the application without injecting code, thus executing sequences of machine instructions (i. e.: gadgets). Usually, each gadget ends in a return instruction and is placed in a subroutine of the program, so it is possible to create a chain of gadgets. The ROP method needs a malicious payload in the application upon installation, unlike our malware model. Differently, we propose a novel Android malware model that uses a series of features inherited from the Android programming language able to load at the run-time of the malicious payload, by hiding the malicious payload by exploiting steganography.

Concluding, nowadays, malware is a topic of research. Milosevic *et al.* in (Milosevic et al., 2016) report a detailed overview of security incidents involving IoT devices from a software viewpoint, showing the most widespread types of malware and exhibiting different types of side-channel attacks.

6 CONCLUSION AND FUTURE WORKS

In this paper, we presented a novel malware model based on a dynamic compilation, dynamic load, and reflection using steganographic techniques. We develop the proposed malware model in the *StegWare* Android application. Moreover, we designed and developed a software framework that makes this new type of malicious payload easy to use and distribute.

We conducted two different experimental analyses, to demonstrate that the *StegWare* application can perpetrate its malicious behavior undisturbed: in fact, the Android's security mechanism does not detect it. Moreover, we also submitted *StegWare* to 73 different free and commercial antimalware who considered it a legitimate application.

Below we reported many solutions to prevent the malicious behavior implemented in the *StegWare* malware model.

The first one consists of notifying the user of all the suspicious events that could threaten the security and privacy of the device owner. A solution at coarse grain is that the device administrator is informed of all the information pieces that the device sends to an external location (a server, another device, a recipient, and so on). In our example, the malware sends private and sensitive information: in this case, the device owner should have received a warning containing all the information that is gathered. Of course, as this mechanism could degrade usability, the user could explicitly specify which kind of information or actions must be considered private or related to security concerns: the user will be informed only when that information is sent somewhere or those actions are performed by an app.

In future work, we will experiment with the possibility to consider audio as a carrier to transmit the malicious payload source code, by applying steganography to an audio file. It will be of interest to understand whether there is the possibility to deliver the malicious payload for instance by listening to audio from streaming.

ACKNOWLEDGEMENT

This work has been partially supported by EU DUCA, EU CyberSecPro, and EU E-CORRIDOR projects and PNRR SERICS_SPOKE1_DISE, RdS 2022-2024 cybersecurity.

REFERENCES

Canfora, G., Mercaldo, F., Moriano, G., and Visaggio, C. A. (2015). Composition-malware: building android malware at run time. In *2015 10th International Conference on Availability, Reliability and Security*, pages 318–326. IEEE.

Casolare, R., Lacava, G., Martinelli, F., Mercaldo, F., Russodivito, M., and Santone, A. (2021). 2faces: a new model of malware based on dynamic compiling and reflection. *Journal of Computer Virology and Hacking Techniques*, pages 1–16.

Johnson, N. F. and Jajodia, S. (1998). Exploring steganography: Seeing the unseen. *Computer*, 31(2):26–34.

Mercaldo., F., Casolare., R., Ciaramella., G., Iadarola., G., Martinelli., F., Ranieri., F., and Santone., A. (2022). A real-time method for can bus intrusion detection by means of supervised machine learning. In *Proceedings of the 19th International Conference on Security and Cryptography - SECRYPT,*, pages 534–539. INSTICC, SciTePress.

Milosevic, J., Sklavos, N., and Koutsikou, K. (2016). Malware in iot software and hardware.

Prandini, M. and Ramilli, M. (2012). Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87.

Scott, J. (2017). Signature based malware detection is dead. *Institute for Critical Infrastructure Technology*.

Siper, A., Farley, R., and Lombardo, C. (2005). The rise of steganography. *Proceedings of student/faculty research day, CSIS, Pace University*.

Wang, T., Lu, K., Lu, L., Chung, S., and Lee, W. (2013). Jekyll on ios: When benign apps become evil. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 559–572.